Treebeard : An Optimizing Compiler for Tree Based ML Inference

ABSTRACT

Decision tree ensembles are commonly used machine learning models generated by machine learning techniques like gradient boosting and random forests. These models are used in a wide range of applications and are deployed at scale (run repeatedly on billions of data items). Several libraries such as XGBoost, LightGBM, and Sklearn expose algorithms for both training and inference with decision tree ensembles. While these libraries incorporate a limited set of hardware-specific optimizations, they do not specialize the inference code to the model being used, leaving significant performance on the table.

This paper presents a compiler-based approach that automatically generates efficient code for decision tree inference. It develops Treebeard, an extensible compiler, which progressively lowers inference code to LLVM IR through multiple intermediate abstractions. By applying model-specific optimizations at the higher levels, loop optimizations at the middle level, and machine-specific optimizations lower down, Treebeard can specialize inference code for each model on each supported hardware target. To improve model inference performance, Treebeard performs several novel optimizations such as tree tiling, tree walk unrolling, and tree walk interleaving.

We implement Treebeard using the MLIR compiler infrastructure and demonstrate the utility of Treebeard by evaluating it on a diverse set of tree ensemble models. Experimental evaluation demonstrates that Treebeard optimizations improve average latency over a batch of inputs by 2.2X over our baseline version. Further, Treebeard is significantly faster than XGBoost and Treelite in both single-core (2.8X and 5.1X respectively) and multi-core (3.2X and 2.6X respectively) settings.

1. INTRODUCTION

Decision tree ensembles are one of the most popular classes of machine learning models [1, 20]. They are generated by machine learning techniques like gradient boosting and random forests. The Kaggle state of data science and machine learning survey [1] shows that these are the most widely used classes of models among data scientists. An analysis of several machine learning pipelines that are in production use in a real large scale web company showed that gradient boosting machines and random forests were the two most widely used ML algorithms in the company [20]. Not only are decision tree based models widely used, they are also used in a diverse range of applications [15]. GBM models were used in the CERN large hadron collider to classify particles based on the data collected from the collider [16]. Other applications include search engines [10], prediction of the financial performance of companies [11], medical diagnostics [6, 24] and recommendation and notification systems [19].

To compute the prediction of a decision tree on an input row x[1], a path from the root to leaf is computed. Each node of a decision tree has a threshold value v and a feature index (or column index) i. At a node, if $x_i < v$, then the walk moves to the left child. If not, we move to the right child. When a leaf is reached, the value of the leaf is returned as the prediction of the tree. Most decision tree based techniques combine several trees into a forest (or ensemble) to improve prediction accuracy. To compute the prediction of the forest, the prediction of each tree is computed and these predictions are then combined (usually by adding them).

Given the wide spread use of decision tree ensembles, the ability to perform high throughput and low latency inference is important. TODO AP Since we don't talk about latency or throughput explicitly, should we be saying this? However, optimizing the tree walks required for decision tree inference on modern CPUs is not straight-forward. Firstly, naively implemented tree walks have poor spatial and temporal locality and therefore poor cache performance [13, 22]. Additionally, since the decision of which node to evaluate next can only be made after evaluating the current node, true dependencies between instructions cause several pipeline stalls. Also, using SIMD instructions to accelerate tree walks is extremely challenging [13].

Several systems like XGBoost [8], LightGBM [14] and Sklearn [2] implement decision tree ensemble inference. These are libraries and implement some optimizations. However, these optimizations are hand coded and need to be manually rewritten or redesigned for every target that is supported. Also, these systems cannot apply specialize inference code depending on the model. A compiler based approach is needed to address these problems. However, existing compilers only support fixed code generation techniques [3, 18]. To fill the need for an extensible compiler infrastructure for decision forest ensembles, we design and build TREEBEARD[2], an optimizing compiler for decision tree ensemble inference. Our motivations for building TREEBEARD are as follows.

[1] An input row is a vector of numbers.

[2] In J. R. R. Tolkein's The Lord of the Rings, Treebeard was the oldest of the Ents left in Middle-earth, an ancient tree-like being who was a "shepherd of trees".

1. Existing libraries [3, 5, 8, 14] perform specific optimizations and are hard to maintain as hardware evolves. Additionally, they cannot tailor inference code to the specific model being used. Past work has identified that the optimizations required change with the model and architectural parameters like cache size [5, 12, 25].
2. The repeated effort to optimize and maintain libraries on several targets is prohibitive. The proliferation of new architectures further exacerbates this problem. Compilers have been successful in alleviating this problem in other domains [9, 21]. However, no such compiler infrastructure currently exists for decision tree ensembles.
3. Currently, no system exists that allows a thorough exploration of the optimization space for decision tree ensemble inference. Compilers have been successful in addressing this problem with ML models like DNNs [4, 7, 9]. However, compiler techniques for decision tree ensembles are less well studied.

We make the following contributions.

1. We design and build TREEBEARD, an extensible compiler infrastructure for decision tree model inference. The infrastructure is built to allow exploration of optimization and code generation techniques. TODO Is this claim too grand?
2. We develop a general infrastructure for the vectorization of decision tree walks based on grouping tree nodes into "tiles". This includes general support for code generation and the in-memory representation of tiled trees. The infrastructure can be used to tile trees based on different cost functions.
3. We show that trees can be tiled using different cost functions. We present two novel tiling methods that are implemented using the general tiling infrastructure.
4. We design and implement various model and loop transformations that significantly improve generated inference code performance to show the power of the proposed framework.

2. TREEBEARD OVERVIEW

Figure 1 shows the high level structure of TREEBEARD. The input to TREEBEARD is a serialized decision tree ensemble. Given an input model our compiler generates optimized inference code. Specifically it generates a callable batch inference function `predictForest` that, given an array of input rows, computes an array of model predictions.

TREEBEARD specializes the code generated for inference by progressively optimizing and lowering a high level representation of the `predictForest` function down to LLVM IR [17]. ***Lowering*** is the process of transforming an operation at a higher abstraction to a sequence of operations at a lower abstraction. Optimizations in TREEBEARD are implemented using a combination of annotation and lowering. An operation at a higher abstraction is annotated with attributes that indicate what kind of optimization is to be performed while lowering it. The lowering transformation uses this information to generate optimized lower level code. For example, tree tiling and loop nest structure (figure 2 shows two possible loop nest structures – one tree at a time and one row at a time) are decided at the highest abstraction. These decisions are communicated to the lowering pass which explicitly encodes them in the lowered IR as shown in Figure 2.

TREEBEARD's IR has three abstraction levels as shown in Figure 2. At the highest level (HIR), the input model is represented as a collection of binary trees. This is shown in the second row of Figure 2. At this level of abstraction, TREEBEARD tiles nodes together to transform a binary tree into an n-ary tree and decides what order trees are to be traversed in. In the example in the figure, trees are tiled with a tile size of 2. Tiling is indicated using colored elipses drawn around nodes that are in the same tile. Tree reordering is another transformation that is performed at this abstraction. One objective of tree reordering is to group identically structured trees so that they can share the same traversal code to improve locality in the instruction cache. Alternative objectives for reordering can easily be supported. With the tiling shown in Figure 2, Tree1 and Tree3 have depth 1, while Tree2 has a depth of 2. Therefore, the trees are reordered so that Tree1 and Tree3 are together.

The high level IR is then lowered to a mid-level IR (MIR). The aim of MIR is to allow optimizations that are independent of the final memory layout of the model. The lowering to MIR performs loop transformations like loop tiling, permutation, parallelization and unrolling. At this level of the IR (the third row in Figure 2), the order in which trees, input row pairs are walked is explicitly represented in the IR using loop nests. Figure 2 shows two possible ways that loop nests could be generated – the first walks all rows for one tree before moving to the next tree while the second walks all trees for one row before moving to the next row. Another aspect handled by this lowering is the fissing of loops to make sure that trees with the same structure share code. Here, both versions of MIR shown ensure that Tree1 and Tree3 share the same traversal code, while traversal code for Tree2 is different. Additionally, tree walk optimizations, such as tree walk unrolling (shown in figure 2) and peeling are performed on the MIR.

TREEBEARD then further lowers the IR to explicitly represent the memory layout of the model. Buffers to hold model values are inserted into the generated code and all tree operations in the mid-level IR are lowered to explicitly reference these buffers. Additionally, at this level of the IR, TREEBEARD generates vectorized code to take advantage of SIMD instructions where applicable. Finally, the inference function is translated to LLVM IR and JIT compiled to executable code.

3. HIGH LEVEL IR OPTIMIZATIONS

This section describes tiling and tree ordering, two optimizations performed at the highest level of abstraction. Recall that at this level the `predictForest` operator is abstractly represented by just a set of trees.

3.1 Notation

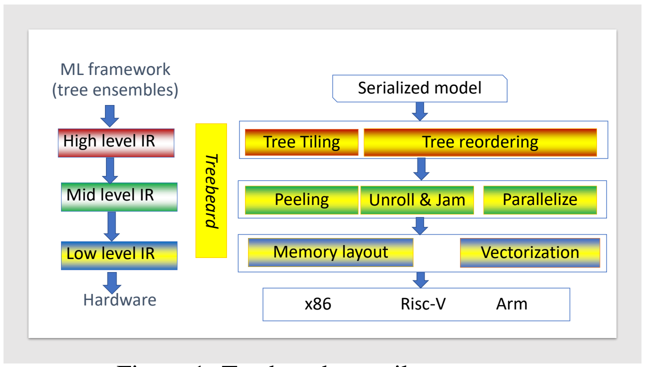


Figure 1: Treebeard compiler structure

TODO Notation needs to be introduced in the background section, Tiling should be the first subsection We represent a decision tree by $T = (V, E, r)$ where V is the set of nodes, E the set of edges and $r \in V$ is the root. For each node $n \in V$, we define the following.

1. $threshold(n) \in \mathbb{R}$, the threshold value for n.
2. $featureIndex(n) \in \mathbb{N}$, the feature index for n.
3. $left(n) \in V$, the left child of n or $\emptyset$ if n is a leaf. If $left(n) \neq \emptyset$, then $(n, left(n)) \in E$.
4. $right(n) \in V$, the right child of n or $\emptyset$ if n is a leaf. If $right(n) \neq \emptyset$, then $(n, right(n)) \in E$.

We use $L \subseteq V$ to denote the set of leaves.

3.2 Tiling

While a decision tree is naturally represented by a binary tree, it is not the best representation for tree traversal as it (i) requires many memory accesses, (ii) has poor branching structure and (iii) cannot make use of vector instructions. This section proposes a tiling optimization where we group multiple nodes in a decision tree into a single tile, effectively transforming a binary tree into an n-ary tiled tree. This not only allows the compiler to generate vectorized code (see Section 5.1) to traverse trees but also enables spatial locality improvements by grouping nodes that are likely to be accessed together. We demonstrate this with two tiling heuristics later in the section.

Once trees are tiled Treebeard generates tree walks with the code structure shown below.

```
ResultType Prediction_Function(...) {
  // ...
  Tile t = getRootTile(tree)
  while (!isLeaf(tree, t)) do {
    // Evaluate conditions of all nodes in the tile
    predicates = evaluateTilePredicates(t, rows[i])

    // Move to the correct child of the current tile
    t = getChildTile(tree, t, predicates)
  }
  treePrediction = getLeafValue(t)
  // ...
}
```

The code is just an abstract representation of a tiled tree walk that enables efficient lowering of specific steps in subsequent stages. `evaluateTilePredicates` (speculatively) computes the predicates on all nodes in a tile (line 6). Then `getChildTile` (line 9), uses the computed predicate values to determine which child of the current tile to move to. We defer a description of how these operators are lowered to a later section but focus on tiling algorithms in this section.

Conditions for Efficient Tiling

While any arbitrary partitioning of the nodes of a tree could be considered for tiling we impose a few intuitive constraints. TODO kr : replace n_t with sz? Given a $tree = (V, E, r)$ and a tile size n_t we impose the following constraints on the generated tiles $\{T_1, T_2, ..., T_m\}$.

Partitioning $T_1 \cup T_2 ... \cup T_m = V$ and $T_i \cap T_j = \emptyset$ for all $i \neq j$

Connectedness If $u, v \in T_i$, there is a (undirected) path connecting u and v fully contained in T_i.

Leaf seperation $\forall l \in L : l \in T_i \rightarrow v \notin T_i \;\; \forall v \in V \setminus \{l\}$

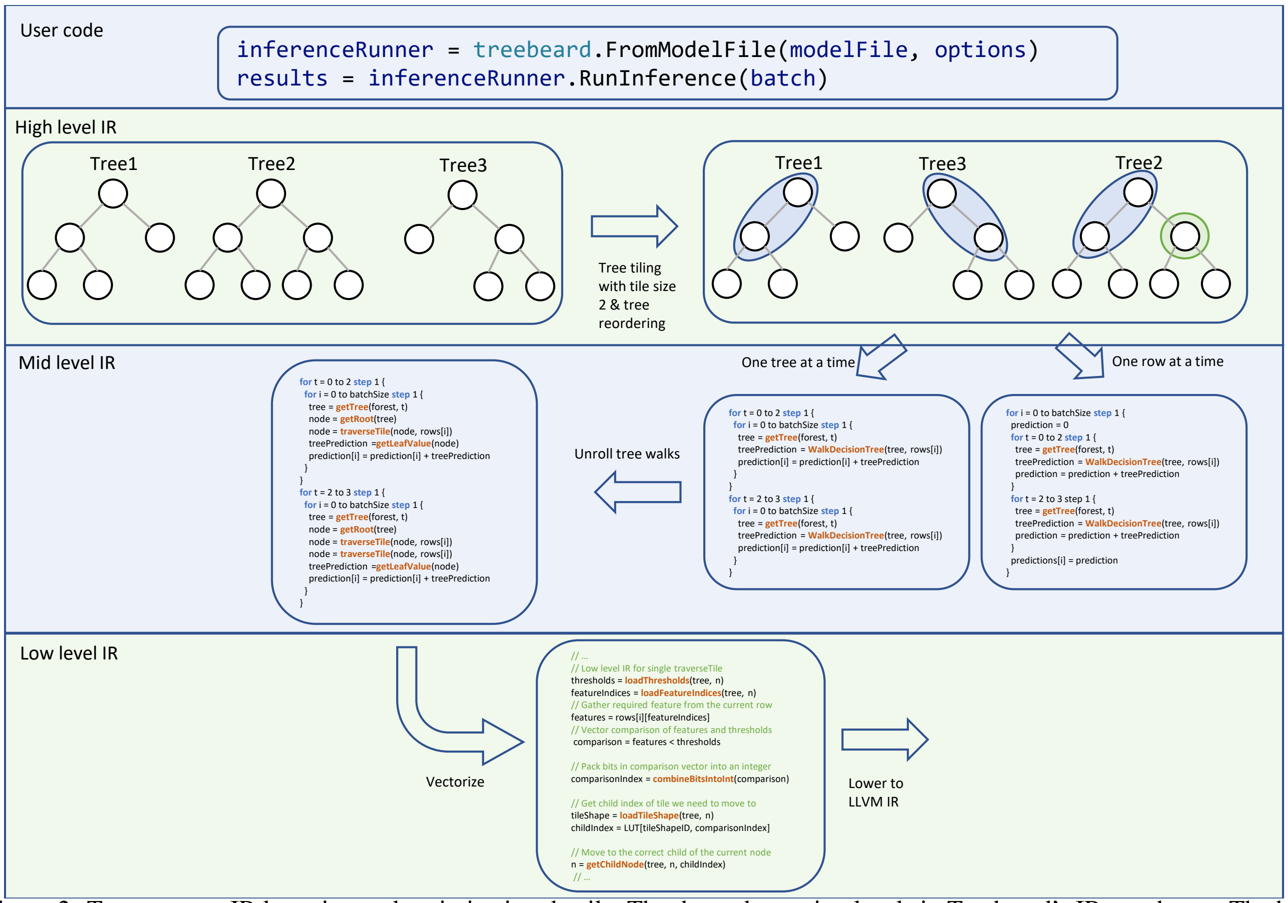
Maximal tiling if there are tiles such that. $|T_i| < n_t$ then there is no $v \in V \setminus \{T_i \cup L\}$ such that $(u, v) \in E$ for some $u \in T_i$.

The **partitioning** and **maximal tiling** constraints together ensure that we group nodes into as few tiles as possible. **Connectedness** ensures that each tile is a sub-tree, a natural grouping of nodes that are likely to be accessed together. The **Leaf separation** constraint ensures that leafs are not tiled along with internal nodes. Leafs in a decision tree need special handling, they are used to check for walk termination and to determine the output (prediction). This constraint ensures that tiles are homogenous, this in-turn allows us to specialize the in-memory layout of trees and also simplifies code generation. We discuss leaf handling and tree layout in Section **??**. We refer to any tiling that satisties the above constraints as a *valid* tilling.

3.3 Basic Tiling Algorithm

Algorithm 1 shows a simple tiling algorithm that satisfies the above constraints and produces a valid tiling. Tiling starts at the root and constructs a tile *Tile* by performing a level order traversal. The call `LevelOrderTraversal(r,` n_t`)` picks the next n_t nodes according to the standard level order tree traversal algorithm (not described here) applied only to internal nodes of the tree. Once the current tile is constructed, the tiling procedure is recursively performed on all nodes that are destinations for edges going out of the constructed tile. It is easy to see that the tiled tree constructed by algorithm 1 is valid. TODO Pass a modified tree without leafs to the level order traversal call. Explicitly add leaves as seperate tiles

One interesting property of this tilling algorithm is that it naturally reduces the imbalance in trees, especially at large tile sizes. As the algorithm traverses down to sparser levels of the tree, it naturally groups sub-trees containing chains of nodes, thus balancing the trees. While its possible to further enhance the algorithm to explicitly balance tiled trees, we find that basic tiling suffices in practice.



```
inferenceRunner = treebeard.FromModelFile(modelFile, options)
results = inferenceRunner.RunInference(batch)
```

```
for t = 0 to 2 step 1 {
 for i = 0 to batchSize step 1 {
  tree = getTree(forest, t)
  node = getRoot(tree)
  node = traverseTile(node, rows[i])
  treePrediction =getLeafValue(node)
  prediction[i] = prediction[i] + treePrediction
 }
}
for t = 2 to 3 step 1 {
 for i = 0 to batchSize step 1 {
  tree = getTree(forest, t)
  node = getRoot(tree)
  node = traverseTile(node, rows[i])
  node = traverseTile(node, rows[i])
  treePrediction =getLeafValue(node)
  prediction[i] = prediction[i] + treePrediction
 }
}
```

```
for t = 0 to 2 step 1 {
 for i = 0 to batchSize step 1 {
  tree = getTree(forest, t)
  treePrediction = WalkDecisionTree(tree, rows[i])
  prediction[i] = prediction[i] + treePrediction
 }
}
for t = 2 to 3 step 1 {
 for i = 0 to batchSize step 1 {
  tree = getTree(forest, t)
  treePrediction = WalkDecisionTree(tree, rows[i])
  prediction[i] = prediction[i] + treePrediction
 }
}
```

```
for i = 0 to batchSize step 1 {
 prediction = 0
 for t = 0 to 2 step 1 {
  tree = getTree(forest, t)
  treePrediction = WalkDecisionTree(tree, rows[i])
  prediction = prediction + treePrediction
 }
 for t = 2 to 3 step 1 {
  tree = getTree(forest, t)
  treePrediction = WalkDecisionTree(tree, rows[i])
  prediction = prediction + treePrediction
 }
 predictions[i] = prediction
}
```

```
// ...
// Low level IR for single traverseTile
thresholds = loadThresholds(tree, n)
featureIndices = loadFeatureIndices(tree, n)
// Gather required feature from the current row
features = rows[i][featureIndices]
// Vector comparison of features and thresholds
 comparison = features < thresholds

// Pack bits in comparison vector into an integer
comparisonIndex = combineBitsIntoInt(comparison)

// Get child index of tile we need to move to
tileShape = loadTileShape(tree, n)
childIndex = LUT[tileShapeID, comparisonIndex]

// Move to the correct child of the current node
n = getChildNode(tree, n, childIndex)
 // ...
```

Figure 2: TREEBEARD IR lowering and optimization details. The three abstraction levels in Treebeard's IR are shown. The high level IR is a tree based IR to perform model level optimization, the mid-level IR is for loop optimizations that are independent of memory layout and the low level IR allows us to perform vectorization and other memory layout dependent optimizations.

Algorithm 1 Uniform tree tiling

```
procedure TILETREE($tree = (V, E, r)$, $n_t$)
    if $r \in L$ then
        return $\{r\}$
    end if
    //Level order traversal to collect $n_t$ or fewer nodes.
    //Leaves are not included in the constructed tile.
    $Tile \leftarrow LevelOrderTraveral(r, n_t)$
    $Tiles = \{Tile\}$
    for $(u, v) \in Out(Tile)$ do
        $Tiles \leftarrow Tiles \cup TileTree(S_v, n_t)$
    end for
    return $Tiles$
end procedure
```

3.4 Probability Based Tiling

The next algorithm we propose exploits the inherent biases among the leaves of a decision tree. In typical machine learning models some leafs (equivalently outcomes or predictions) are more likely to be reached than others. In such settings, having balanced tiled trees is not sufficient to minimize expected inference time.

Consider for example two machine learning models `airline-ohe` and `epsilon` (also used in our evaluation). Consider the graphs shown in figures 3 and 4 that are generated from training data. Each line in these graphs corresponds to a fixed fraction of the input (say f). A point on a line at coordinate (x, y) means that a fraction y of trees in the model could cover a fraction f of all training inputs with a fraction x of leaves. For example, the first point on the $f = 0.9$ line in figure 3 says that about 52% of trees (y value) need only 1% of their leaves (x value) to cover 90% of the training input. In general, Figure 3 shows that very few leaves are needed to cover a very large fraction of inputs for the benchmark `airline-ohe`. This means that a small fraction of leaves are very likely. We call trees with a small number of extremely likely leaves ***leaf biased***.

On the other hand, for the benchmark `epsilon`, figure 4 shows that a trees need a much larger fraction of their leaves to cover a significant fraction of the training input. This means that most trees in `epsilon` are not leaf biased.

3.4.1 *The Optimization Problem*

Observe that the latency of one tree walk is proportional to the number of tiles that need to be evaluated to reach the leaf. It is easy to see that for a leaf biased tree, basic tilling does not optimize for this objective, it considers all leafs to be equally likely.

The goal of probablistic tiling is to minimize the average inference latency, or equivalently the minimize the expected number of tiles that are evaluated to compute one tree prediction. More formally, the problem is to find a *valid* (as defined in Section 3.2) tiling $\mathscr{T}$ such that the following objective is minimized.

$$\min_{\mathscr{T} \in \mathscr{C}(T)} \sum_{l \in L} p_l . depth_{\mathscr{T}}(l)$$

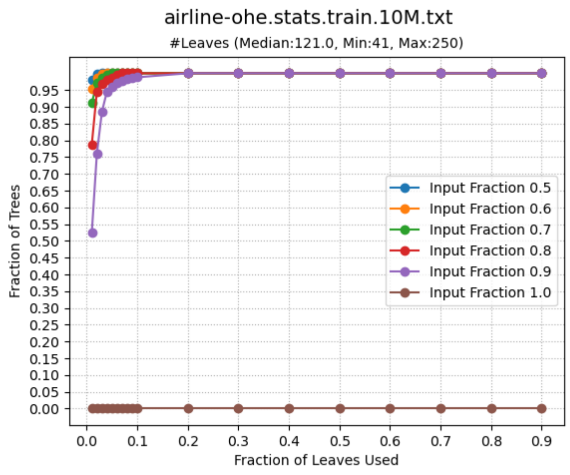


Figure 3: Statistical profile for airline-ohe

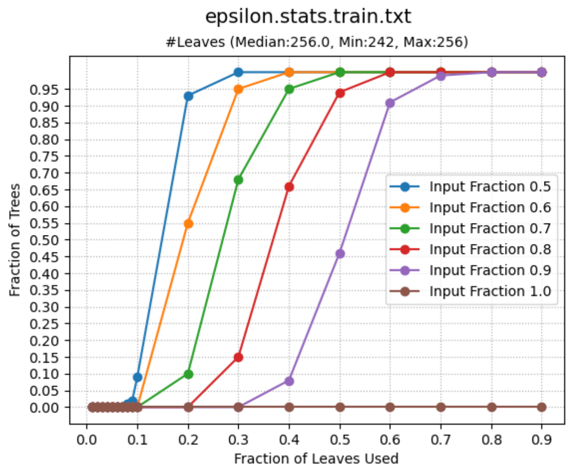


Figure 4: Statistical profile for epsilon

where the minimization is over all valid tilings $\mathscr{T}$ of the tree T, $depth_{\mathscr{T}}(l)$ is the depth of the leaf l given tiling $\mathscr{T}$. p_l is the probability of of reaching leaf l as observed during training.

The above optimization problem can be solved optimally using dynamic programming. We leave this out in the interest of space. Instead, we use the simple greedy algorithm listed in algorithm 2 to construct a valid tiling given the node probabilities[3]. The algorithm starts at the root and greedily keeps adding the most probable legal node to the current tile until the maximum tile size is reached. Subsequently, the tiling procedure is recursively performed on all nodes that are destinations for edges going out of the constructed tile.

Algorithm 2 Greedy Probability Based Tree Tiling

```
procedure TILETREE($T = (V, E, r)$, $n_t$)
    if $r \in L$ then
        return $\{r\}$
    end if
    $Tile \leftarrow \{r\}$
    while $|Tile| < n_t$ do
        $e = (u, v) \in Out(Tile)$ st $p(v)$ is max and $v \notin L$
        if $e = \emptyset$ then
            break
        end if
        $Tile = Tile \cup \{v\}$
    end while
    $Tiles = \{Tile\}$
    for $(u, v) \in Out(Tile)$ do
        $Tiles \leftarrow Tiles \cup TileTree(S_v, n_t)$
    end for
    return $Tiles$
end procedure
```

We find probability based tiling is only beneficial for leaf biased trees[4]. Recall that a tree to be leaf biased if a small fraction of leaves, say α, can cover a large fraction of training inputs, say β. We only perform probability based tiling on trees with thresholds $\alpha = 0.05$ and $\beta = 0.9$ and fall back to uniform tiling otherwise.

3.5 A note on implementation

TODO Kr : Not sure if this is needed The tiling algorithms generate `TileId` attributes per tree and this information is used when lowering to the mid level abstraction in the form of loops. The `TileId` attribute maintains a mapping from a Node to the TileId asigned to it. A sample of the lowered MIR code is shown in Figure 2.

3.6 Tree ordering

TODO kr : Not sure if i got this para right Specializing the code for each tree in a model comes at a cost. First the code generator needs to generate different code for different trees potentially increasing the size of the generated code. Second some cross tree optimizations (applied at the lower levels of abstraction) like tree walk interleaving require that the multiple trees share identical code.

In order to handle this, TREEBEARD pads trees with dummy nodes to make them balanced and then sorts the trees by their depth, so that trees of same depth can share code. Padding is only done for almost balanced trees (as generated by basic tiling), this is ensured by only adding up to a fixed fraction of dummy nodes. TODO Kr : Can we add a threshold say 10%? Once this is done, the loop over the trees is fissed so that each of the resulting loops only walks trees of a single depth. Consider for example a forest with 4 trees T_1, T_2, T_3, and T_4 in that order. Further, assume that T_1 and T_4 have depth 2 while T_2 and T_3 have depth 3. First, Treebeard reorders the trees to be in the order T_1, T_4, T_2, T_3. Then, the loop over the trees is fissed as shown in the following listing.

```
forest = ensemble(...)
for i = 0 to batchSize step 1 {
  prediction = 0
  for t = 0 to 2 step 1 {
    tree = getTree(forest, t)
    node = getRoot(tree)
    node = traverseTreeTile(tree, node, rows[i])
    treePrediction = getLeafValue(tree, node)
    prediction = prediction + treePrediction
  }
  for t = 2 to 4 step 1 {
    tree = getTree(forest, t)
    node = getRoot(tree)
    node = traverseTreeTile(tree, node, rows[i])
    node = traverseTreeTile(tree, node, rows[i])
    treePrediction = getLeafValue(tree, node)
    prediction = prediction + treePrediction
  }
  predictions[i] = prediction
}
```

4. LOOP OPTIMIZATIONS AT MID LEVEL

4.1 Tree Walk Interleaving

While tiling and subsequent vectorization gives significant performance gains, profiling the generated code showed that true dependencies between instructions were still causing a significant number of stall cycles. In order to fill these stall cycles, Treebeard does an unroll and jam on the inner most loops of the loop nest. This transformation has the effect of walking multiple tree and input row pairs in an interleaved fashion. The dependency stalls can be hidden by scheduling instructions from independent tree walks in the stall cycles.

This optimization is performed across both the mid-level IR and low-level IR. Loops on which to perform unroll and jam are identified in the mid-level IR. This information is communicated to the lowering passes by annotating these tree walk operations as descibred in section **??**. The lowering passes that transform the mid-level IR to low-level IR interleave operations across independent tree walks.

The following listing shows the mid-level IR when the inner loop over the input rows is unrolled by a factor of 2 and the two resulting tree walks are jammed together.

[3]Probabilites for internal nodes can be computed from probablities for leafs by summing up the probabilities of all leafs that belong to the sub-tree rooted at the internal node. Leaf probabilities are collected during training.

[4]Turns out that for trees that are not leaf biased, probability based tiling produces many more tile shapes (see Section **??**) which direclty impacts the cost of `getChildTile` making it more expensive than basic tiling.

```
forest = ensemble(...)
for t = 0 to numTrees step 1 {
  for i = 0 to batchSize step 2 {
    tree = getTree(forest, t)
    prediction1, prediction2 = InterleavedWalk((tree,
   rows[i]), (tree, rows[i+1]))
  }
}
```

When lowered to low-level IR, the operations to traverse each of the tree, input row pairs (the arguments to the `InterleavedWalk`) are interleaved. One step of the interleaved walk is listed below.

```
// ...
n1 = n2 = getRoot(tree)
// ...
threshold1 = loadThresholds(n1)
threshold2 = loadThresholds(n2)
featureIndex1 = loadFeatureIndices(n1)
featureIndex2 = loadFeatureIndices(n2)
feature1 = rows[i][featureIndex1]
feature2 = rows[i][featureIndex2]
pred1 = feature1 < threshold1
pred2 = feature2 < threshold2
n1 = getChildNode(n1, pred1)
n2 = getChildNode(n2, pred2)
// ...
```

These transformations are fairly general and are not aware of the in memory representation of the model. Therefore, they are reusable across different in memory representations - the ones that are currently built into Treebeard or ones that maybe added in the future. TODO AP I feel the way this section is currently written makes the optimization seem extremely trivial. Is there a different way to present it?

4.2 Loop peeling and walk unrolling

TREEBEARD splits the loop that performs a tree walk into two parts. As it is aware of the depth of the first leaf in a tree, it peels and introduces a `prologue` loop that walks down the tree a constant number of steps (up to first leaf) and then performs the rest of the tree walk in a separate loop. Further TREEBEARD unrolls the prologue completely, avoiding all traversal induced branching in it.

Note that for trees where TREEBEARD has already padded and balanced the tree (Section 3.6), walk unrolling completely avoids all traversal induced spills. TODO kr: Add example?

4.3 Parallelization

Currently, Treebeard performs a naive parallelization of the inference computation. When parallelism is enabled, the loop over the input rows is parallelized using OpenMP. Treebeard rewrites the mid-level IR by tiling the loop over the input rows with a tile size equal to the number of cores. As a concrete example, consider the case where we intend to perform inference using a model with 4 trees on a batch of 64 rows. Further, assume that we wish to parallelize this computation across 8 cores. Treebeard then generates the following IR.

```
forest = ensemble(...)
parallel.for i0 = 0 to 64 step 8 {
  for i1 = 0 to 8 step 1 {
    i = i0 + i1
    prediction = 0
    for t = 0 to 4 step 1 {
      tree = getTree(forest, t)
      treePrediction = WalkTree(tree, rows[i])
      prediction = prediction + treePrediction
    }
    predictions[i] = prediction
  }
}
```

Currently, as our focus is on single core performance, we do not do any parallelism related optimizations (such as loop tiling). We leave a more thorough exploration of parallelization to future work.

5. LOW LEVEL OPTIMIZATIONS

5.1 Vectorization

Vectorization performed by Treebeard is enabled by the tiling transformations described in section 3.2. When the low level IR is translated to LLVM IR, Treebeard generates LLVM instructions that operate on the threholds and feature indices of nodes within a tile in a vector fashion. Therefore, thresholds and feature indices are loaded using vector loads and predicates are evaluated using vector comparisons. These vector LLVM IR instructions are then converted to vector instructions in the target ISA by the LLVM JIT.

The below listing shows some of the details of a vectorized tree walk.

```
1  // A lookup table that determines the child index of
2  // the next tile given the tile shape and the outcome
3  // of the vector comparison on the current tile
4  int16_t LUT[NUM_TILE_SHAPES, pow(2, TileSize)]
5
6  ResultType Prediction_Function(...) {
7    // ...
8    Node n = getRoot(tree)
9    while (isLeaf(tree, n)==false) do {
10     thresholds = loadThresholds(tree, n)
11     featureIndices = loadFeatureIndices(tree, n)
12     // Gather required feature from the current row
13     features = rows[i][featureIndices]
14     // Vector comparison of features and thresholds
15     comparison = features < thresholds
16
17     // Pack bits in comparison vector into an integer
18     comparisonIndex = combineBitsIntoInt(comparison)
19
20     // Get child index of tile we need to move to
21     tileShape = loadTileShape(tree, n)
22     childIndex = LUT[tileShapeID, comparisonIndex]
23
24     // Move to the correct child of the current node
25     n = getChildNode(tree, n, childIndex)
26   }
27   ThresholdType prediction = getLeafValue(n)
28   // ...
29 }
```

To evaluate the current tile, the vector of thresholds is first loaded (`loadThresholds`). This vector contains the thresholds of all nodes in the tile. Then, the features required for comparison are gathered into a vector (lines 11 and 13). The feature vector is compared to the threshold vector and the child tile to move to is determined (lines 15 to 25). More details about tile shapes and the look up table are presented in subsequent sections.

5.1.1 Tile Shapes

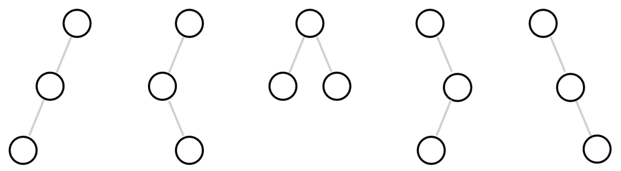


Figure 5: All possible tile shapes with tile size $n_t = 3$

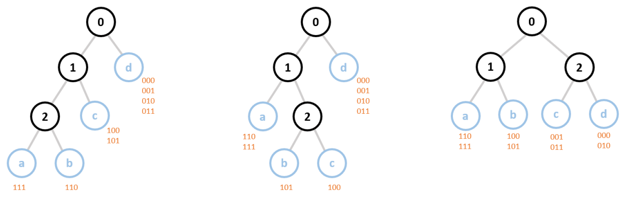


Figure 6: Example tile traversals with tile size $n_t = 3$

Informally, the ***tile shape*** is the shape of the region that encloses all nodes in a tile in a diagram of the decision tree. More formally, for a tile size n_t, each unique legal binary tree containing n_t nodes (nodes being indistinguishable) corresponds to a tile shape.

Figure 5 enumerates all tile shapes with a tile size of 3. There are a total of 5 tile shapes with size 3. The number of tile shapes with a tile size n_t, denoted by $NTS(n_t)$ is given by the following equation.

$$NTS(n) = \sum_{k=0}^{n-1} NTS(k) \times NTS(n-k-1) \qquad (1)$$

where $NTS(0) = NTS(1) = 1$.

5.1.2 Tile Shapes and Decision Tree Inference

Treebeard uses vector instructions to accelerate decision tree walks. Vector instructions are used to evaluate the predicates of all the nodes in a tile simultaneously. However, once the predicates of all the nodes in the tile are evaluated, computing the next tile to move to, given the outcome of the comparison depends on the tile shape of the current tile. To illustrate this problem, consider the case of the tiles of size 3 shown in figure 6. The diagram shows 3 of the 5 possible tile shapes for a tile size of 3. The nodes drawn in black are members of the tile t_1. The nodes in blue are the entry nodes of the children tiles of t_1. TODO Define entry nodes

In the diagram, the bit strings (written in red) show which child we need to move to given the outcomes of the comparison. The bits represent the comparison outcomes of nodes and are in the order of the nodes in the tile – marked 0, 1 and 2 in the diagram, i.e., the MSB is the predicate outcome of node 0 and the LSB the predicate outcome of node 2. For example, for the first tile shape, if the predicates of all nodes are true (i.e. the comparison outcome is 111), the next node to evaluate is a. It is easy to see from the diagram that, depending on the tile shape, the same predicate outcomes can mean moving to different children. For example, for the outcome "011", the next tile is the 4th child (node d) for the first two tile shapes while it is the 3rd child for the other tile shape (node c).

5.1.3 Lookup Table

A lookup table (LUT) is used to solve the problem described in section 5.1.2, i.e. given the outcome of the comparisons of all nodes in a tile, determine the child tile we should evaluate next. The LUT is indexed by the tile shape and the comparison outcome. Formally, the LUT is a map.

$$LUT : (TileShape, < Boolean \times n_t >) \rightarrow [0, n_t] \subset \mathbb{N}$$

where n_t is the tile size, $< Boolean \times n_t >$ is a vector of n_t booleans. The value returned by the LUT is the index of the child of the current tile that should be evaluated next. For example, if we are evaluating the first tile t in figure 6, and the result of the comparison is 110, then $LUT(TileShape(t), 110) = 1$ since the tile we need to evaluate next is the tile with node b, which is the second child of the current tile.

In order to realize this LUT in generated code, Treebeard associates a non-negative integer ID with every unique tile shape of the given tile size. The result of the comparison, a vector of booleans, can be interpreted as a 64-bit integer. Therefore, the LUT can be implemented as a 2 dimensional array. Treebeard computes the values in the LUT statically as the tile size is a compile time constant. TODO AP What comes after subsubsection?

5.2 In Memory Representation of Tiled Trees

Treebeard currently has two in memory representations for tiled trees - an array based representation and a sparse representation. Both representations use an array of structs to represent all tiles of the model.

5.2.1 Array Based Representation

Each tree in the model is represented as an array of tiles using the standard representation of trees as arrays. The root node is at index 0 and for a node at index n in the array, the index of its i^{th} child is given by $(n_t + 1)n + (i + 1)$ (nodes in the tree of tiles have $n_t + 1$ children). A tile is represented by an object of the following struct.

```
struct Tile {
  // A vector of TileSize elements
  <ThresholdType x TileSize> thresholds;
  <FeatureIndexType x TileSize> featureIndices;
  // Integer that identifies the tile shape
  TileShapeIDType tileShapeID;
};
```

TODO AP Is this level of detail really needed? Also, the vector type notation needs to be introduced somewhere. Even though this representation is simple and efficient for small models, the memory required for bigger models is very large. This memory bloat causes performance problems because the span of the L1 TLB is not sufficient to efficiently translate addresses for the whole model. Storing leaves as full tiles (even though leaves just have to represent one value) and the empty space introduced due to the array based representation of trees that are not complete account for most of the increase.

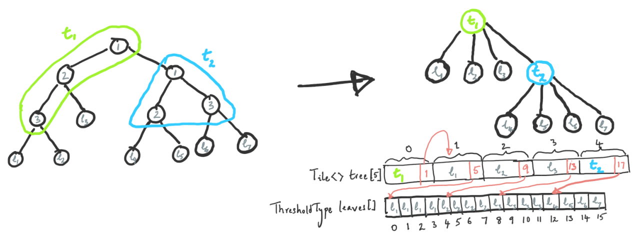


Figure 7: Sparse representation with tile size $n_t = 3$

5.2.2 *Sparse Representation*

The sparse representation tries to address the large memory footprint of the array based representation by doing the following.

- We add a child pointer to each tile to eliminate the wasted space in the array representation. This points to the first child of the tile. All children of a tile are stored contiguously.

- Leaves are stored as a separate array of scalar values. Across all our benchmarks, after tiling a large fraction of leaves are such that all their siblings are also leaves. Such leaves are directly moved into the leaves array. For leaves for which this property does not hold, an extra "hop" is added by making the original leaf tile a normal tile. All its children are made leaves with the same value as the original leaf.

Figure 7 shows some of the details of the sparse representation. The arrays depicted below show how the tree is represented in memory. The first array (`tree`) is an array of tiles and has 5 elements. Each element of the array represents a single tile and has the thresholds of the nodes, the feature indices, a tile shape ID and a pointer to the first child (shown explicitly in red).

As a specific example, consider the tile t_1. The tile has four children – l_1, l_2, l_3 and t_2 in that order (left to right). These tiles are stored contiguously in the `tree` array and a pointer to the first of these, l_1 is stored in the tile t_1 (the index 1 is stored in the tile t_1 as shown).

Now consider the tile t_2. Since all children of the tile t_2 are leaves, they are all moved into the `leaves` array. To store a pointer into the `leaves` array, we add `len(tree)` to the element index in the `leaves` array. The tile t_2's child is the element at index 12 of the `leaves` array. Therefore, the index $12+5=17$ is stored in the tile t_2. Any index i that is greater than the length of the `tree` array is regarded as an index into the `leaves` array. The index into the `leaves` array is $i-$`len(tree)`.

The other aspect of the representation is that an extra hop is added for the leaves l_1, l_2 and l_3 in order to simplify code generation. This enforces the invariant that all leaves are stored in the leaves array and simplifies checking whether we've reached a leaf. Therefore, 4 new leaves are added as children for each of the original leaves l_1, l_2 and l_3. Each of these 12 newly added leaves has the same value as its parent. These are the first 12 elements of the `leaves` array.

Even though we currently have implementations of the two representations detailed in sections 5.2.1 and 5.2.2, support for other representations is not hard to add. All optimizing passes that work on the high level and mid level IR will continue to work as is. Programmers need only provide new lowering passes for a few operations in the low level IR.

5.2.3 *Code Generation for Probability Based Tiling*

As probability based tiling pulls the most probable leaves of a decision tree nearest the root, it poses some implementation challenges. By design, the tiling process makes the tree of tiles imbalanced. The array based representation (section 5.2.1) cannot be used because of the memory footprint increase (a large part of the tree is empty, but would need to be allocated). On the other hand, the sparse representation in section 5.2.2 adds an extra hop for leaves that have non-leaf siblings. But this would mean that we add extra hops for the most probable leaves after probability based tiling which defeats the optimization. TODO kr: connect to Peeling in MIR We address these challenges using a code generation strategy. Treebeard peels the tree walk and specializes the leaf checks at higher levels to avoid the extra hop. Currently, we determine the maximum depth of leaves needed to cover 90 percent of the inputs and peel the tree walk by as many iterations. For example, consider the case where leaves until depth 2 are needed to cover 90 percent of the training input. Then, Treebeard generates the following IR.

```
    // ...
    tree = getTree(forest, t)
    node = getRoot(tree)
    node = traverseTreeTile(tree, node, rows[i])
    if (isLeafTile(node)) {
        treePrediction = getLeafValue(tree, node)
    } else {
        node = traverseTreeTile(tree, node, rows[i])
        if (isLeafTile(node)) {
            treePrediction = getLeafValue(tree, node)
        } else {
            // Loop based traversal
        }
    }
    treePrediction = getLeafValue(tree, node)
    // ...
```

The if statements check whether a node is a leaf tile and hence avoid the extra hop. While walk peeling is used to improve the performance of probability based tiling by specializing leaf tests, the transformation is by itself general and can be used in different contexts. For example, it could be used to elide leaf checks until a depth d is reached if we know all leaves are at a depth greater than d.

One other issue that the code generator needs to handle is that walks of different trees in the same ensemble may need to be peeled to different depths. A strategy similar to what is used for uniform tiling is used to handle this. Trees are reordered so that all trees with equal peeling depth are grouped together and the loops in the IR are fissed so that tree walks for these groups of trees can be specialized differently.

6. RELATED WORK

Decision Tree Ensemble Compilers: Treelite [3] only supports generation of if-else style code and is not extensible.

Hummingbird [18] tries to compile decision tree models to tensor primitives so they can be integrated into tensor based frameworks like TensorFlow [**?**].

Libraries: Currently, the most popular systems for decision tree based models are libraries. XGBoost [8] and LightGBM [14] are the most popular gradient boosting libraries while scikit-learn [2] is extremely popular for random forest models. These libraries implement both training and inference.

Other Systems and Techniques: Ren et al [23] build an intermediate language and VM to enable SIMD execution of decision tree inference. The SIMD execution itself is implemented by hand in the VM and the VM needs to be reimplemented for every supported target architecture. Additionally, even though they perform layout optimizations, their system doesn't perform any model specific optimizations.

Tahoe, Vpred, CacheConscious1, QuickScorer, QuickScorer1, MilindTreeVectorization Bliss?

Decision tree training parallelization references

REFERENCES

[1] "Kaggle state of data science and machine learning 2021," https://www.kaggle.com/kaggle-survey-2021, accessed: 2022-04-16.

[2] "scikit-learn : Machine learning in python," https://scikit-learn.org/stable/, accessed: 2022-04-16.

[3] "Treelite : model compiler for decision tree ensembles," https://treelite.readthedocs.io/en/latest/, accessed: 2022-04-16.

[4] "Xla : Optimizing compiler for machine learning," https://www.tensorflow.org/xla, accessed: 2022-04-16.

[5] N. Asadi, J. Lin, and A. P. de Vries, "Runtime optimizations for tree-based machine learning models," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2281–2292, 2014.

[6] A. Azar and S. El-Metwally, "Decision tree classifiers for automated medical diagnosis," *Neural Computing and Applications*, vol. 23, pp. 2387–2403, 11 2013.

[7] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 193–205.

[8] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[10] D. Cossock and T. Zhang, "Statistical analysis of bayes optimal subset ranking," *IEEE Transactions on Information Theory*, vol. 54, no. 11, pp. 5140–5154, 2008.

[11] D. Delen, C. Kuzey, and A. Uyar, "Measuring firm performance using financial ratios: A decision tree approach," *Expert Systems with Applications*, vol. 40, p. 3970–3983, 08 2013.

[12] X. Jin, T. Yang, and X. Tang, "A comparison of cache blocking methods for fast execution of ensemble-based score computation," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 629–638. [Online]. Available: https://doi.org/10.1145/2911451.2911520

[13] Y. Jo, M. Goldfarb, and M. Kulkarni, "Automatic vectorization of tree traversals," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. IEEE Press, 2013, p. 363–374.

[14] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 3149–3157.

[15] S. Kotsiantis, "Decision trees: A recent overview," *Artificial Intelligence Review*, pp. 1–23, 04 2013.

[16] V. Lalchand, "Extracting more from boosted decision trees: A high energy physics case study," 2020. [Online]. Available: https://arxiv.org/abs/2001.06033

[17] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[18] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi, "A tensor compiler for unified machine learning prediction serving," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 899–917. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/nakandala

[19] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. S. Khudia, J. Law, P. Malani, A. Malevich, N. Satish, J. M. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. M. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *CoRR*, vol. abs/1811.09886, 2018. [Online]. Available: http://arxiv.org/abs/1811.09886

[20] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer, "Data science through the looking glass and what we found there," 2019. [Online]. Available: https://arxiv.org/abs/1912.09536

[21] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[22] B. Ren, T. Mytkowicz, and G. Agrawal, "A portable optimization engine for accelerating irregular data-traversal applications on simd architectures," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, jun 2014. [Online]. Available: https://doi.org/10.1145/2632215

[23] B. Ren, T. Mytkowicz, and G. Agrawal, "A portable optimization engine for accelerating irregular data-traversal applications on simd architectures," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, jun 2014. [Online]. Available: https://doi.org/10.1145/2632215

[24] J. Soni, U. Ansari, D. Sharma, and S. Soni, "Predictive data mining for medical diagnosis: An overview of heart disease prediction," *International Journal of Computer Applications*, vol. 17, pp. 43–48, 03 2011.

[25] X. Tang, X. Jin, and T. Yang, "Cache-conscious runtime optimization for ranking ensembles," in *Proceedings of the 37th International ACM SIGIR Conference on Research amp; Development in Information Retrieval*, ser. SIGIR '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1123–1126. [Online]. Available: https://doi.org/10.1145/2600428.2609525